

Restaurante

Desenvolvimento do trabalho 2 de Sistemas Operativos



Trabalho realizado por:

Gabriel Martins Silva (113786)
Filipe Ramalho de Oliveira (114640)

Ano Letivo: [2023/2024]

Índice

Introdução.....	3
Chef.....	4
Função waitForOrder().....	4
Função processOrder().....	5
Waiter.....	6
Função waitForClientOrChef().....	6
Função informChef().....	8
static void informChef (int n).....	8
Função takeFoodToTable().....	9
Receptionist.....	11
Função decideTableOrWait().....	11
Função decideNextGroup().....	12
Função waitForGroup().....	12
Função provideTableOrWaitingRoom().....	14
Função receivePayment().....	15
Group.....	17
Função checkInAtReception().....	17
Função orderFood().....	18
Função waitFood().....	19
Função checkOutAtReception().....	20
Testes e validação.....	22
Estados de Group (G).....	24
Estados de Chef (CH).....	24
Estados de Waiter (WT).....	25
Estados de Receptionist (RC).....	25
Conclusão.....	25

Introdução

O nosso trabalho trata-se do completamento de quatro ficheiros .c, de forma a simular o funcionamento de um restaurante. Este restaurante tem duas mesas disponíveis, um cozinheiro (chef), um empregado de mesa (waiter), um recepcionista (receptionist) e diferentes grupos de clientes (group).

Essencialmente, o chef espera por pedidos de comida, processa esses pedidos, e comunica-se com o waiter para entregar a comida pronta.

O waiter lida com os pedidos de comida, comunicação com o chef e entrega de comida às mesas.

O receptionist espera por solicitações de grupos, decide se um grupo deve ser alocado a uma mesa ou esperar, e processa pagamentos, liberando mesas para outros grupos.

O group todas as ações desde a chegada ao restaurante até a saída, passando por pedir comida, esperar pela comida, comer e pagar a conta.

A manipulação de semáforos é uma parte crucial nesta implementação, garantindo a sincronização correta entre as diferentes entidades referidas.

Chef

Função `waitForOrder()`

```
static void waitForOrder ()
{
    // Wait for a food order from the waiter
    if (semDown(semgid, sh->waitOrder) == -1) {
        perror("error on the down operation for wait order semaphore (PT)");
        exit(EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    lastGroup = sh->fSt.foodGroup; // Save the group that requested food

    // Update chef's state to COOK
    sh->fSt.st.chefStat = COOK;
    saveState(nFic, &sh->fSt); // Save the state

    // Acknowledge the received order
    if (semUp(semgid, sh->orderReceived) == -1) {
        perror("error on the up operation for order received semaphore
(PT)");
        exit(EXIT_FAILURE);
    }

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
}
```

Numa fase inicial aguarda um pedido de comida que será fornecido pelo waiter, o chef entra num estado de espera sendo o semáforo *waitOrder* utilizado para a sincronização entre o chef e o waiter. De seguida, o programa entra na região crítica e, mal este recebe um pedido, o seu estado é alterado para *COOK*, simulando assim o tempo em que o chef estaria a cozinhar. O pedido é reconhecido através da operação *semup* no semáforo

orderReceived, ao executar esta operação o chefe será libertado para começar a processar o pedido, após isto o programa sai da região crítica.

Função processOrder()

```
static void processOrder ()
{
    // Simulate cooking time
    usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));

    // Wait for waiter to be available
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the up operation for chef semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    // request waiter to deliver food
    sh->fSt.waiterRequest.reqGroup = lastGroup;
    sh->fSt.waiterRequest.reqType = FOODREADY;

    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for chef semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    // Update chef's state to WAIT_FOR_ORDER
    sh->fSt.st.chefStat = WAIT_FOR_ORDER;
    saveState(nFic, &sh->fSt); // Save the state

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }
}
```

O objetivo desta função é simular o processo de confecção do pedido dado ao chef.

A função começa por definir o tempo que demora este processo, através do uso da função *usleep*. O tempo é calculado aleatoriamente através da função *random*.

Antes de sinalizar que a comida está pronta, o chef precisa garantir que o waiter esteja disponível para receber a comida. Isso é feito através da chamada *semDown* no semáforo *waiterRequestPossible*.

O programa entra na região crítica e o chef atualiza a requisição para o waiter, definindo *reqGroup* para *lastGroup* e *reqType* para *FOODREADY*, indicando o estado em que a comida está pronta.

Em seguida, o chef realiza uma operação *semup* no semáforo *waiterRequest*, sinalizando o waiter de que a comida para o grupo *lastGroup* está pronta.

Após completar essa tarefa o chef atualiza o seu próprio estado para *WAIT_FOR_ORDER*, ficando assim pronto para esperar um novo pedido e o programa sai da região crítica.

Waiter

Função `waitForClientOrChef()`

```
static request waitForClientOrChef()
{
    request req;
    bool foundRequest = false;

    if (semDown(semgid, sh->mutex) == -1) {          /* enter critical region */
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    // Update waiter's state to WAIT_FOR_REQUEST and save the state
    sh->fSt.st.waiterStat = WAIT_FOR_REQUEST;
    saveState(nFic, &sh->fSt);

    if (semUp(semgid, sh->mutex) == -1) {             /* exit critical region */
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    // Signal readiness for new requests
    if (semUp(semgid, sh->waiterRequestPossible) == -1) {
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }
}
```

```

// Wait for a request from a group or chef
while (!foundRequest) {
    if (semDown(semgid, sh->mutex) == -1) { /* enter critical region */
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    // Check for group food requests
    for (int i = 0; i < MAXGROUPS; i++) {
        if (sh->fSt.st.groupStat[i] == FOOD_REQUEST) {
            req.reqType = FOODREQ;
            req.reqGroup = i;
            foundRequest = true;
            sh->fSt.st.groupStat[i] = WAIT_FOR_FOOD;
            break;
        }
    }

    // Check for chef's food ready signal
    if (!foundRequest && sh->fSt.waiterRequest.reqType == FOODREADY) {
        for (int i = 0; i < MAXGROUPS; i++) {
            if (sh->fSt.st.groupStat[i] == WAIT_FOR_FOOD) {
                req.reqType = FOODREADY;
                req.reqGroup = i;
                foundRequest = true;
                break;
            }
        }
    }

    if (semUp(semgid, sh->mutex) == -1) { /* exit critical region */
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }
}

return req;
}

```

Esta função é responsável por esperar uma solicitação de um cliente ou do chef. O programa entra na região crítica e o waiter atualiza o seu estado para *WAIT_FOR_REQUEST*, mostrando assim que ele está pronto a receber um pedido. Com a operação *semup* no *mutex* o programa volta a entrar em região crítica. O waiter sinaliza a sua disponibilidade para receber solicitações através da operação *semup* no semáforo *waiterRequestPossible*.

O programa entra num *while loop* enquanto uma solicitação válida não for encontrada e verifica se algum grupo fez um pedido de comida. Se um pedido for encontrado, o tipo de solicitação *reqType* é definido como *FOODREQ*, o grupo é salvo, a solicitação é marcada como encontrada, e o status do grupo é atualizado para *WAIT_FOR_FOOD*. Se nenhum pedido for encontrado, o programa verifica se o chef sinalizou que a comida está pronta. Caso isso seja verdade, o tipo de solicitação será definido como *FOODREADY*, o grupo é salvo e o programa sai da região crítica.

Função `informChef()`

```
static void informChef (int n)
{
    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Update waiter's state to INFORM_CHEF and save the state
    sh->fSt.st.waiterStat = INFORM_CHEF;
    saveState(nFic, &sh->fSt);

    // Set request type and group ID for the chef
    sh->fSt.waiterRequest.reqType = COOK; // Request chef to cook
    sh->fSt.waiterRequest.reqGroup = n;

    // Signal the chef that a request has been made
    if (semUp(semgid, sh->waitOrder) == -1) {
        perror("error on the up operation for chef request semaphore (WT)");
        exit(EXIT_FAILURE);
    }

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Wait for chef to acknowledge the request
    if (semDown(semgid, sh->orderReceived) == -1) {
        perror("error on the down operation for chef response semaphore
(WT)");
        exit(EXIT_FAILURE);
    }
}
```



```

// Get the table number from the request
int tableId = sh->fSt.assignedTable[n];

// Signal the group that the request has been received
if (semUp(semgid, sh->requestReceived[tableId]) == -1) {
    perror("error on the up operation for request semaphore (WT)");
    exit(EXIT_FAILURE);
}
}

```

Inicialmente, o programa entra em uma região crítica. Uma vez na região crítica, o waiter atualiza seu estado para *INFORM_CHEF*, refletindo que ele está no processo de informar o chef sobre um novo pedido de comida. O programa define os detalhes do pedido e especifica o *reqType* como *COOK*.

Após atualizar a solicitação, o waiter executa a operação *semUp* no semáforo *waitOrder*. Esta ação sinaliza ao *chef* que um novo pedido foi feito e que ele pode começar o processo de preparação da comida.

Por fim, o programa sai da região crítica.

Função takeFoodToTable()

```

static void takeFoodToTable(int n)
{
    if (semDown(semgid, sh->mutex) == -1) {          /* enter critical region */
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    sh->fSt.st.waiterStat = TAKE_TO_TABLE;
    saveState(nFic, &sh->fSt);

    // Get the table number from the request
    int tableId = sh->fSt.assignedTable[n];

    // Signal the group that food is ready
    if (semUp(semgid, sh->foodArrived[tableId]) == -1) {
        perror("error on the up operation for food arrived semaphore (WT)");
        exit(EXIT_FAILURE);
    }

    if (semUp(semgid, sh->mutex) == -1) {            /* exit critical region */
        perror("error on the up operation for semaphore access (WT)");
    }
}

```

```

        exit(EXIT_FAILURE);
    }

    if (semDown(semgid, sh->mutex) == -1) {          /* enter critical region */
        perror("error on the down operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }

    sh->fSt.st.groupStat[n] = EAT;  // Update group's state to EAT
    saveState(nFic, &sh->fSt);

    if (semUp(semgid, sh->mutex) == -1) {            /* exit critical region */
        perror("error on the up operation for semaphore access (WT)");
        exit(EXIT_FAILURE);
    }
}

```

A principal finalidade da função *takeFoodToTable* é levar a comida, já preparada pelo chef, até a mesa do grupo correspondente.

No início da função, o programa entra novamente em uma região crítica através da operação *semDown* no mutex.

Dentro da região crítica, o estado do waiter é atualizado para *TAKE_TO_TABLE*, um indicativo claro de que ele está no processo de entrega da comida.

O waiter então identifica o grupo específico para o qual a comida deve ser entregue e após a identificação do grupo, o waiter realiza a operação *semUp* no semáforo correspondente ao grupo, indicando que a comida está a caminho e que o grupo pode se preparar para receber o pedido. Finalmente o programa sai da região crítica.

Receptionist

Função *decideTableOrWait()*

```
static int decideTableOrWait(int n)
{
    // Ensure the group is at the reception
    if (sh->fSt.st.groupStat[n] != ATRECEPTION) {
        return -1;
    }

    // Iterate through each table to check if it is occupied
    for (int tableId = 0; tableId < NUMTABLES; tableId++) {
        bool isOccupied = false;

        // Check if the current tableId is assigned to any group
        for (int groupId = 0; groupId < sh->fSt.nGroups; groupId++) {
            if (sh->fSt.assignedTable[groupId] == tableId) {
                isOccupied = true;
                break; // This table is already occupied
            }
        }

        // If the table is not occupied, return its ID
        if (!isOccupied) {
            return tableId;
        }
    }

    return -1; // All tables are occupied, so the group must wait
}
```

A função *decideTableOrWait* determina se uma mesa está disponível a receber clientes ou não.

A função começa verificando se o grupo especificado está na recepção. Em seguida, a função inicia um *for loop* que percorre cada mesa disponível no restaurante verificando se a mesa está ocupada através de um segundo *for loop* que verifica se algum grupo já foi alocado à mesa em questão. Se o programa encontrar uma mesa não ocupada a função retorna imediatamente o id dessa mesa, caso contrário retorna -1.

Função decideNextGroup()

```
static int decideNextGroup()
{
    for (int groupId = 0; groupId < sh->fSt.nGroups; groupId++) {
        if (decideTableOrWait(groupId) != -1 && groupRecord[groupId] == WAIT)
        {
            return groupId;
        }
    }

    return -1; // Return the group ID or -1 if no group is waiting
}
```

A função *decideNextGroup* no sistema do receptionist tem o objetivo de determinar qual grupo será o próximo a ser alocado a uma mesa.

A função inicia com um ciclo que percorre todos os grupo, dentro deste ciclo a função referida anteriormente (*decideTableOrWait*) é chamada para cada grupo. Caso um grupo esteja em um estado de espera (*WAIT*), a função retorna o ID do mesmo, caso contrário retorna -1.

Função waitForGroup()

```
static request waitForGroup()
{
    request ret;
    if (semDown (semgid, sh->mutex) == -1) { /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Update receptionist status to WAIT_FOR_REQUEST and save the state
    sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) { /* exit critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Wait for a group to make a request
    if (semDown(semgid, sh->receptionistReq) == -1) {
        perror("error on the down operation for receptionist semaphore access (WT)");
    }
}
```

```

        exit(EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {        /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Copy the request details from shared memory
    ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
    ret.reqType = sh->fSt.receptionistRequest.reqType;

    // Signal that the receptionist is ready for new requests
    if (semUp (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the up operation for receptionist semaphore access
(WT)");
        exit (EXIT_FAILURE);
    }

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    return ret;
}

```

A função *waitForGroup* lida com a espera e o processamento de solicitações dos grupos de clientes.

A função começa por colocar o programa em região crítica. Dentro desta, o receptionist atualiza seu estado para *WAIT_FOR_REQUEST*, estando a aguardar uma solicitação de um grupo.

Após atualizar o estado, o receptionist realiza *semUp* no *mutex*, saindo da região crítica.

A função entra em uma espera ativa pela solicitação de um grupo através de *semDown* no semáforo *receptionistReq*. O programa volta a entrar em região crítica e, dentro da mesma, o receptionist copia os detalhes da solicitação do grupo da memória compartilhada para a estrutura “*ret*”, que inclui o identificador do grupo e o tipo de solicitação. Através de *semUp* no semáforo *receptionistRequestPossible*, o receptionist indica a sua disponibilidade de atender mais grupos.

Função provideTableOrWaitingRoom()

```
static void provideTableOrWaitingRoom (int n)
{
    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Check if the group is arriving for the first time
    if (groupRecord[n] == TOARRIVE) {

        // Decide if the group can be assigned a table or must wait
        int tableId = decideTableOrWait(n);

        if (tableId != -1) {
            // If a table is available

            // Update receptionist status to ASSIGNTABLE and save the state
            sh->fSt.st.receptionistStat = ASSIGNTABLE;
            saveState(nFic, &sh->fSt);

            // Assign the table to the group
            sh->fSt.assignedTable[n] = tableId;

            // Signal the group that it can proceed to the table
            if (semUp(semgid, sh->waitForTable[n]) == -1) {
                perror("error on the up operation for group wait for table
semaphore (RT)");
                exit(EXIT_FAILURE);
            }

            groupRecord[n] = ATTABLE; // Update internal receptionist view

        } else {
            // If the group must wait
            groupRecord[n] = WAIT; // Update internal receptionist view

            sh->fSt.groupsWaiting++; // Update the number of groups waiting
        }

    }

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the down operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }
}
```

A função *provideTableOrWaitingRoom* é responsável por determinar se um grupo específico pode ser imediatamente alocado a uma mesa ou deve aguardar na sala de espera.

Inicialmente, o programa entra em uma região crítica, a função verifica se o grupo está a chegar pela primeira vez, e caso esteja a função *decideTableOrWait* é chamada para avaliar a disponibilidade de uma mesa para o grupo. Se uma mesa estiver disponível o receptionist atualiza seu estado para *ASSIGNTABLE*. O receptionist então aloca a mesa ao grupo e sinaliza o grupo que pode prosseguir para a mesa atribuída através de *semUp* no semáforo *waitForTable*. O estado interno do receptionist é atualizado para *ATABLE*, indicando que o grupo foi alocado a uma mesa. Caso não haja mesas disponíveis, o grupo é colocado em estado de espera (*WAIT*) e o contador de grupos à espera é incrementado.

Função *receivePayment()*

```
static void receivePayment (int n)
{
    if (semDown (semgid, sh->mutex) == -1) {        /* enter critical region */
        perror ("error on the up operation for semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Update receptionist state to receiving payment and save the state
    sh->fSt.st.receptionistStat = RECVPAY;
    saveState(nFic, &sh->fSt);

    // Identify the table being vacated
    int tableId = sh->fSt.assignedTable[n];

    if (semUp (semgid, sh->tableDone[tableId]) == -1) {
        perror ("error on the down operation for receptionist semaphore access (WT)");
        exit (EXIT_FAILURE);
    }

    // Update the internal receptionist view to indicate the group is done
    groupRecord[n] = DONE;
    sh->fSt.assignedTable[n] = -1; // Mark the table as vacant

    // Check if there are waiting groups
    if (sh->fSt.groupsWaiting > 0) {

        // Decide which group will be assigned next
        int nextGroup = decideNextGroup();
```

```

    if(nextGroup != -1){
        // If there is a group waiting

        // Assign the table to the group
        sh->fSt.assignedTable[nextGroup] = tableId;
        groupRecord[nextGroup] = ATTABLE;

        // Signal the group that it can proceed to the table
        if (semUp(semgid, sh->waitForTable[nextGroup]) == -1) {
            perror("error on the up operation for group wait for table
semaphore (RT)");
            exit(EXIT_FAILURE);
        }

        // Decrease the number of groups waiting
        sh->fSt.groupsWaiting--;
    }
}

if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
    perror ("error on the down operation for semaphore access (WT)");
    exit (EXIT_FAILURE);
}
}

```

A função *receivePayment* é encarregada de gerenciar o pagamento e a liberação de mesas no restaurante.

A função inicia entrando em região crítica onde o receptionist atualiza seu estado para *RECVPAY*. O receptionist identifica a mesa que está a ser libertada pelo grupo, obtendo o seu *tableId*. Após identificar a mesa, o receptionist realiza *semUp* no semáforo *tableDone[tableId]*, indicando que a mesa foi levantada e está pronta para ser realocada. O receptionist atualiza então o registo do grupo para *DONE*, indicando que o grupo concluiu o pagamento.

Após isso a função verifica se há algum grupo à espera de mesa e caso haja, a função *decideNextGroup* é chamada para determinar qual grupo irá para a mesa a seguir. Havendo um grupo em espera o registo do grupo é atualizado para *ATTABLE* sendo sinalizado, através de *semUp* no semáforo *waitForTable[nextGroup]*, que se pode dirigir à mesa e o contador de grupos em espera é decrementado.

Group

Função checkInAtReception()

```
static void checkInAtReception(int id)
{
    // Wait until the receptionist is ready to take a request
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Update group status to ATRECEPTION and save state
    sh->fSt.st.groupStat[id] = ATRECEPTION;
    saveState(nFic, &sh->fSt);

    // Prepare and send table request to receptionist
    sh->fSt.receptionistRequest.reqType = TABLEREQ;
    sh->fSt.receptionistRequest.reqGroup = id;

    // Signal receptionist that a new request has been made
    if (semUp (semgid, sh->receptionistReq) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Wait for a table to be assigned
    if (semDown (semgid, sh->waitForTable[id]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}
```

A função *checkInAtReception* tem o intuito de registrar a sua chegada e solicitar uma mesa. A função inicia com a espera pela disponibilidade do receptionist para receber uma solicitação, utilizando a operação *semDown* no semáforo *receptionistRequestPossible*. De seguida o programa entra em região crítica e o estado do grupo é atualizado para *ATRECEPTION*. O grupo prepara então o seu pedido de mesa, atualizando a solicitação do receptionist na estrutura *receptionistRequest* com *TABLEREQ* e o *id*. É usado *semUp* no semáforo *receptionistReq* para notificar o receptionist que um novo pedido foi feito. O programa sai da região crítica e, por fim, o grupo aguarda que uma mesa seja atribuída através de *semDown* no semáforo *waitForTable[id]*.

Função orderFood()

```
static void orderFood (int id)
{
    // Wait until it's possible to make a request to the waiter
    if (semDown (semgid, sh->waiterRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Update group status to FOOD_REQUEST and save state
    sh->fSt.st.groupStat[id] = FOOD_REQUEST;
    saveState(nFic, &sh->fSt);

    // Prepare food request for the waiter
    sh->fSt.waiterRequest.reqType = FOODREQ;
    sh->fSt.waiterRequest.reqGroup = id;

    // Signal waiter that a new food request has been made
    if (semUp (semgid, sh->waiterRequest) == -1) {
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Get assigned table of the group
    int tableId = sh->fSt.assignedTable[id];

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
    // Wait for the waiter to acknowledge the food request
```

```

        if (semDown (semgid, sh->requestReceived[tableId]) == -1) {
            perror ("error on the down operation for semaphore access (CT)");
            exit (EXIT_FAILURE);
        }
    }
}

```

A função *orderFood* centra-se no processo de fazer um pedido de comida ao empregado de mesa.

A função inicia com a espera pela disponibilidade do empregado de mesa para receber pedidos, utilizando a operação *semDown* no semáforo *waiterRequestPossible*.

Quando a disponibilidade é confirmada o programa entra em região crítica onde o estado do grupo é atualizado para *FOOD_REQUEST*.

O grupo prepara o seu pedido atualizando *waiterRequest* com *FOODREQ* e o id e através de *semUp* no semáforo *waiterRequest* o waiter é sinalizado de que um novo pedido de comida foi realizado. O ID da mesa do grupo é obtido e o programa sai da região crítica. Por fim, o grupo aguarda que o waiter reconheça o pedido de comida através de *semDown* no semáforo *requestReceived*.

Função waitFood()

```

static void waitFood (int id)
{
    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Update group status to WAIT_FOR_FOOD and save state
    sh->fSt.st.groupStat[id] = WAIT_FOR_FOOD;
    saveState(nFic, &sh->fSt);

    // Get assigned table of the group
    int tableId = sh->fSt.assignedTable[id];

    if (semUp (semgid, sh->mutex) == -1) {             /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Wait for the food to arrive
    if (semDown (semgid, sh->foodArrived[tableId]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {           /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
    }
}

```

```

        exit (EXIT_FAILURE);
    }

    // Update group status to EAT and save state
    sh->fSt.st.groupStat[id] = EAT;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}

```

A função *waitFood* é caracterizada pelo processo de espera pela chegada da comida após fazerem o pedido.

Entrando em região crítica, o estado do grupo é atualizado para *WAIT_FOR_FOOD* e o programa obtém então o id da mesa atribuída ao grupo.

O programa sai da região crítica, e o grupo é seguido de um período de espera até a comida chegar, utilizando *semDown* no semáforo *foodArrived[tableId]*.

O programa volta a entrar na região crítica assim que a comida chega e o estado do grupo é alterado para *EAT*, estando o grupo a comer.

Função checkOutAtReception()

```

static void checkOutAtReception (int id)
{
    // Wait until the receptionist is ready to process the checkout
    if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Update group status to CHECKOUT and save state
    sh->fSt.st.groupStat[id] = CHECKOUT;
    saveState(nFic, &sh->fSt);

    // Prepare payment request for the receptionist
    sh->fSt.receptionistRequest.reqType = BILLREQ;
    sh->fSt.receptionistRequest.reqGroup = id;

    // Signal receptionist that a new payment request has been made
    if (semUp (semgid, sh->receptionistReq) == -1) {

```

```

        perror ("error on the up operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Get assigned table of the group
    int tableId = sh->fSt.assignedTable[id];

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Wait for the receptionist to acknowledge the payment
    if (semDown (semgid, sh->tableDone[tableId]) == -1) {
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    if (semDown (semgid, sh->mutex) == -1) {          /* enter critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }

    // Update group status to LEAVING and save state
    sh->fSt.st.groupStat[id] = LEAVING;
    saveState(nFic, &sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {          /* exit critical region */
        perror ("error on the down operation for semaphore access (CT)");
        exit (EXIT_FAILURE);
    }
}

```

A função *checkOutAtReception* concentra-se no processo de pagamento e saída do grupo. A função começa com o grupo à espera que o receptionist esteja pronto para processar o pagamento, através de *semDown* no semáforo *receptionistRequestPossible*.

Mal o receptionist se encontrar pronto, o programa entra em região crítica onde o estado do grupo é atualizado para *CHECKOUT*, indicando que o grupo se encontra em processo de pagar.

O grupo faz o seu pedido de pagamento atualizando a estrutura *receptionistRequest* com *BILLREQ* e o ID e usando *semUp* no semáforo *receptionistReq*, notificando assim o receptionist de que o grupo quer fazer o pagamento.

O programa sai da região crítica, e o grupo aguarda que o receptionist reconheça o pedido de pagamento através de *semDown* no semáforo *tableDone[tableId]*.

O programa volta a entrar em região crítica e o estado do grupo é atualizado para *LEAVING*, indicando assim que o grupo se está a retirar do restaurante.

Testes e validação

CH	WT	RC	G00	G01	G02	G03	G04	gWT	T00	T01	T02	T03	T04
0	0	0	1	1	1	1	1	0
0	0	0	1	1	1	1	1	0
0	0	0	1	1	1	1	1	0
0	0	0	1	2	1	1	1	0
0	0	1	1	2	1	1	1	0
0	0	1	1	2	2	1	1	0	.	0	.	.	.
0	0	0	1	2	2	1	1	0	.	0	.	.	.
0	0	0	1	3	2	1	1	0	.	0	.	.	.
0	1	0	1	4	2	1	1	0	.	0	.	.	.
0	1	1	1	4	2	1	1	0	.	0	.	.	.
0	1	0	1	4	2	1	1	0	.	0	1	.	.
0	1	0	1	4	3	1	1	0	.	0	1	.	.
1	1	0	1	4	3	1	1	0	.	0	1	.	.
1	0	0	1	4	3	1	1	0	.	0	1	.	.
1	0	0	1	4	3	1	1	0	.	0	1	.	.
1	1	0	1	4	4	1	1	0	.	0	1	.	.
0	1	0	1	4	4	1	1	0	.	0	1	.	.
1	1	0	1	4	4	1	1	0	.	0	1	.	.
1	0	0	1	4	4	1	1	0	.	0	1	.	.
1	0	0	1	4	4	1	1	0	.	0	1	.	.
1	2	0	1	4	4	1	1	0	.	0	1	.	.
1	2	0	1	5	4	1	1	0	.	0	1	.	.
1	2	0	1	5	4	1	1	0	.	0	1	.	.
1	0	0	1	5	4	1	1	0	.	0	1	.	.
1	2	0	1	5	4	1	1	0	.	0	1	.	.
1	2	0	1	5	5	1	1	0	.	0	1	.	.
1	2	0	1	5	5	1	1	0	.	0	1	.	.
1	0	0	1	5	5	1	1	0	.	0	1	.	.
0	0	0	1	5	5	1	1	0	.	0	1	.	.
0	0	0	1	5	5	2	1	0	.	0	1	.	.
0	0	0	1	5	5	2	1	1	.	0	1	.	.
0	0	0	1	5	5	2	2	1	.	0	1	.	.
0	0	0	1	5	5	2	2	2	.	0	1	.	.
0	0	0	2	5	5	2	2	2	.	0	1	.	.
0	0	0	2	5	5	2	2	3	.	0	1	.	.
0	0	0	2	5	6	2	2	3	.	0	1	.	.
0	0	2	2	5	6	2	2	3	.	0	1	.	.
0	0	0	2	5	6	2	2	2	1	0	.	.	.
0	0	0	2	5	7	2	2	2	1	0	.	.	.
0	0	0	3	5	7	2	2	2	1	0	.	.	.
0	1	0	4	5	7	2	2	2	1	0	.	.	.
1	1	0	4	5	7	2	2	2	1	0	.	.	.
1	0	0	4	5	7	2	2	2	1	0	.	.	.
1	0	0	4	5	7	2	2	2	1	0	.	.	.
0	0	0	4	5	7	2	2	2	1	0	.	.	.

0	2	0	4	5	7	2	2	2	1	0	.	.	.
0	2	0	5	5	7	2	2	2	1	0	.	.	.
0	0	0	5	5	7	2	2	2	1	0	.	.	.
0	0	0	5	5	7	2	2	2	1	0	.	.	.
0	0	0	6	5	7	2	2	2	1	0	.	.	.
0	0	2	6	5	7	2	2	2	1	0	.	.	.
0	0	0	6	5	7	2	2	1	.	0	.	1	.
0	0	0	6	5	7	3	2	1	.	0	.	1	.
0	0	0	7	5	7	3	2	1	.	0	.	1	.
0	1	0	7	5	7	4	2	1	.	0	.	1	.
1	1	0	7	5	7	4	2	1	.	0	.	1	.
1	0	0	7	5	7	4	2	1	.	0	.	1	.
1	0	0	7	5	7	4	2	1	.	0	.	1	.
0	0	0	7	5	7	4	2	1	.	0	.	1	.
0	2	0	7	5	7	4	2	1	.	0	.	1	.
0	2	0	7	5	7	5	2	1	.	0	.	1	.
0	0	0	7	5	7	5	2	1	.	0	.	1	.
0	0	0	7	5	7	5	2	1	.	0	.	1	.
0	0	0	7	5	7	6	2	1	.	0	.	1	.
0	0	2	7	5	7	6	2	1	.	0	.	1	.
0	0	0	7	5	7	6	2	0	.	0	.	.	1
0	0	0	7	5	7	7	2	0	.	0	.	.	1
0	0	0	7	5	7	7	3	0	.	0	.	.	1
0	1	0	7	5	7	7	4	0	.	0	.	.	1
1	1	0	7	5	7	7	4	0	.	0	.	.	1
1	0	0	7	5	7	7	4	0	.	0	.	.	1
1	0	0	7	5	7	7	4	0	.	0	.	.	1
0	0	0	7	5	7	7	4	0	.	0	.	.	1
0	2	0	7	5	7	7	4	0	.	0	.	.	1
0	2	0	7	5	7	7	5	0	.	0	.	.	1
0	2	0	7	5	7	7	6	0	.	0	.	.	1
0	2	2	7	5	7	7	6	0	.	0	.	.	1
0	2	0	7	5	7	7	6	0	.	0	.	.	.
0	2	0	7	5	7	7	7	0	.	0	.	.	.
0	2	0	7	6	7	7	7	0	.	0	.	.	.
0	2	2	7	6	7	7	7	0	.	0	.	.	.
0	2	2	7	7	7	7	7	0

Esta tabela representa o resultado de um teste efetuado ao restaurante implementado pelas funções descritas previamente. As siglas CH, WT e RC são indicativas dos estados do Chef, Waiter e Receptionist, respetivamente. Os diferentes G representam os vários grupos que visitaram o restaurante. A sigla gWT indica o número de grupos à espera de mesa no momento, e T(grupo) mostra a mesa atualmente ocupada por cada grupo. Após a realização de diversos testes com soluções aleatórias ao código, concluímos que este se revelou um sucesso.

Na tabela apresentada, os estados são representações numéricas dos diferentes estágios ou atividades de cada elemento no restaurante durante o teste. Cada número corresponde a um estado específico. Por exemplo, no caso dos clientes, um número pode indicar que o grupo está à espera de uma mesa, a pedir comida, a comer, ou a realizar o pagamento. Estes números são fundamentais para entender o teste, proporcionando-nos uma visão clara sobre a evolução de cada entidade no sistema.

As seguintes tabelas representam os diferentes valores dos estados de cada entidade:

Estados de Group (G)

Estado	Valor	Descrição
GOTOREST	1	Group initial state
ATRECEPTION	2	Group is waiting at reception or for a table
FOOD_REQUEST	3	Group is requesting food to waiter
WAIT_FOR_FOOD	4	Group is waiting for food
EAT	5	Group is eating
CHECKOUT	6	Group is checking out
LEAVING	7	Group is leaving

Estados de Chef (CH)

Estado	Valor	Descrição
WAIT_FOR_ORDER	0	Chef waits for food order
COOK	1	Chef is cooking
REST	2	Chef is resting

Estados de Waiter (WT)

Estado	Valor	Descrição
WAIT_FOR_REQUEST	0	Waiter waits for food request
INFORM_CHEF	1	Waiter takes food request to chef
TAKE_TO_TABLE	2	Waiter takes food to table

Estados de Receptionist (RC)

Estado	Valor	Descrição
ASSIGNTABLE	1	Receptionist assigns table
RECVPAY	2	Receptionist receives payment

Conclusão

Ao longo deste relatório exploramos detalhadamente os diversos componentes e funções que compõem o sistema de simulação do restaurante. O sucesso desta simulação foi o uso eficiente de semáforos, que permitiram uma sincronização entre as diferentes entidades : o chef, o Receptionist, o Waiter e o Group.

Com o trabalho realizado, sentimos que enriquecemos os nossos conhecimentos numa área onde ainda não estávamos muito confortáveis e desta forma poderemos tirar proveito para desafios futuros.